MASTER COPY KEEP THIS COPY FOR REPRODUCTION PURPOSES

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED Reprint
4. TITLE AND SUBTITLE Title shown on Reprint		5. FUNDING NUMBERS DAAL03-91-G-0215
6. AUTHOR(S) Author(s) listed on Reprint		
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Rensselaer Polytechnic Institute Troy, NY 12180-8397		8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) U. S. Army Research Office P. O. Box 12211 Research Triangle Park, NC 27709-2211		10. SPONSORING/MONITORING AGENCY REPORT NUMBER ARO 29167.10-MA
11. SUPPLEMENTARY NOTES The view, opinions and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Army position, policy, or decision, unless so designated by other documentation.		
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited.		12b. DISTRIBUTION CODE
13. ABSTRACT *(Maximum 200 words)* ABSTRACT ON REPRINT		
14. SUBJECT TERMS		15. NUMBER OF PAGES
		16. PRICE CODE

19950203 220

17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. Z39-18
298-102

Scalable Software Tools for Adaptive Scientific Computation*

Boleslaw K. Szymanski, Can Özturan, and Joseph E. Flaherty
Department of Computer Science, Rensselaer Polytechnic Institute
Troy, New York 12180-3590, USA

Abstract

With a constant need to solve scientific and engineering problems of ever growing complexity, there is a corresponding need for software tools that assist in generating solutions with minimal user involvement. Parallel computation is becoming indispensable in solving the large-scale problems that arise in science and engineering applications. Adaptivity is at the center of efficient methods for solving partial differential equations often used in such applications. Yet the use of parallel computation and adaptive techniques is limited by the high cost of developing the needed software. To overcome this difficulty, we advocate a comprehensive approach to the development of scalable architecture-independent software for adaptive solutions of partial differential equations.

Our approach is based on program decomposition, parallel computation synthesis and run-time support for adaptive computations. Parallel program decomposition is guided by the source program annotations provided by the user. A family of annotation languages has been designed for this purpose. The synthesis of parallel application is based on configurations that describe overall computation and interaction of its components. Run-time support is responsible for redistributing data and computation during program execution in response to changing computational needs of different subregions during adaptive solution. Adaptive finite difference and finite element procedures tuned to a specific size and type of parallel architecture will be synthesized from components of a decomposed source programs. In this paper, we discuss annotations and configurations suitable for parallel programs written in FORTRAN or in the functional parallel programming language called EPL.

*This work was partially supported by National Science Foundation under grants CCR-8920694 and CDA-8805910

1 Introduction

Several problems listed as "grand challenges" of the Federal High-Performance Computing Program [16] involve the solution of complex multi-dimensional steady and transient partial differential equations. As the mathematical models include more realistic effects, all of these problems exceed the capabilities of current computer systems. We believe, as others do, that computer performance in the needed range of teraflops can be attained only through massive parallelism. However, raw computing power alone is not sufficient to solve a complex problem. We must ensure that (i) adequate mathematical models are used, (ii) reliable numerical methods are employed to approximate these models, (iii) accurate parallel implementations of the methods are executed, (iv) results are within prescribed numerical accuracy, and (v) parallel implementations use the available computational power efficiently.

Adaptivity, with its associated error estimation and shrewd use of computation only in regions where accuracy requirements are not satisfied, provides the needed numerical reliability and efficiency. Adaptive solutions often converge at rates that are much higher than those obtained by conventional methods using a single grid. At the same time, adaptive methods are challenging from the point of view of programming complexity because they use sophisticated data structures, recursion, run-time domain and method selection, etc. Parallelism adds to this challenge because software development for parallel architectures is more complex than for sequential machines due to the increased complexity of assuring parallel program correctness and efficiency. Parallel program correctness requires the results to be independent of the number and speed of the processors. This scalability requirement can be satisfied only if the parallel tasks are independent of each other or properly synchronized when a dependence exists. Synchronization design and verification are the major source of difficulty in assessing parallel program correctness. Different categories of parallel architectures have led to a proliferation of dialects of standard computer languages. Varying parallel programming primitives for different parallel language dialects greatly limits parallel software portability. Clearly, the large efforts required to develop and implement parallel adaptive solution techniques have hampered their widespread application by scientists and engineers. In addition, poor portability of parallel programs has resulted in duplication of effort and has limited the use of developed systems.

The aims of scientific computation are to further understanding of natural phenomena by implementing and executing mathematical models when experiments would be impractical and/or to supplement experiments when direct measurements are not possible. Large-scale computation requires high performance parallel architectures and efficient program implementation to attain acceptable execution times. To facilitate scientific parallel program development, there is a need for software tools that will support **efficiency** as well as:

scalability – the same mathematical models and numerical algorithms are often used in computations with different accuracy and size and executed with a variable number of processors; hence, the cost of the algorithms used in the software tools should increase slowly with the increase in the number of processors used (e.g. the cost function is poly-logarithmic in the number of processors used),

reusability – basic numerical algorithms frequently appear in different models and different computations,

extensibility – interactive development and stepwise refinement of mathematical models describes an implementation of the new models in terms of changes to the old model.

Design methodology of software tools with the above properties is currently a research goal of great importance. Our approach to such design methodology is based on decomposition and scalable synthesis of parallel programs for scientific and engineering computation. The goal is to enable the users to describe high-level features of a parallel computation and to synthesize computation from numerical algorithms, program fragments, and data structures that are separately implemented. Such decomposition and synthesis can support (i) parallel task formulation and allocation, (ii) data distribution, (iii) run-time optimization, and (iv) rapid prototyping of different parallel implementations.

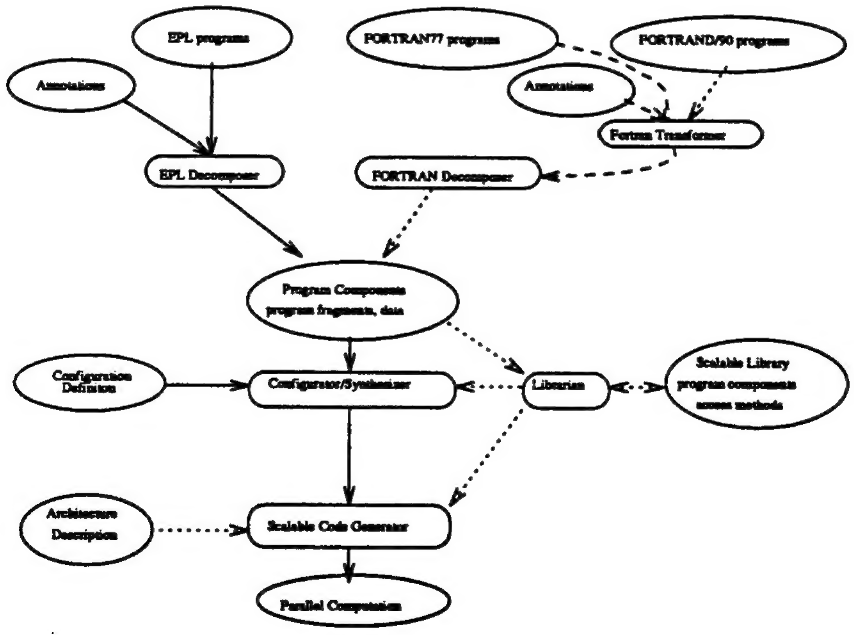

Figure 1: Software tools and their use

The summary view of our approach is given in Figure 1. Program components are created by annotating source programs in FORTRAN or in the functional parallel programming language EPL [14]. FORTRAN programs are transformed into an equational form before decomposition. The configuration definition guides the synthesis of the components into a parallel computation. The synthesized computation together with the architecture description is used by the code generator to produce an object code customized for the target architecture. In the future, we will add a scalable library and an associated librarian to increase versatility of the system. In Figure 1, continuous lines describe implemented paths of the system, broken lines represent paths currently under construction, and dotted lines correspond to paths at an early stage of investigation.

This paper is intended as an overview of the research done towards implementing software tools as envisioned in Figure 1. More technical discussion can be found elsewhere [4, 10, 13, 14, 15].

The paper is organized as follows. Annotations and program decomposition are discussed in Section 2. Program synthesis and the design of the configurator are presented in Section 3. A dynamic load management strategy for adaptive scientific computation on SIMD architectures is a topic of Section 4. Finally, conclusions are outlined in Section 5.

2 Annotations

Annotations provide an efficient way of introducing user's directives for assisting the compiler in parallelization. To be effective, annotations have to be carefully limited to a few constructs. They also should preserve semantics of the original program. In our approach, annotations are introduced solely to limit the allocation of computations to processors. Hence, programs decorated with annotations produce the same results as unannotated program. Consequently, sequential programs that have manifested their correctness over many years of usage are good candidates for parallelization through annotations. By being orthogonal to the program description, annotations support rapid prototyping of different parallel solutions.

2.1 Annotations in EPL

In EPL, each equation can be annotated with the name of the virtual processor on which it is to be executed. Virtual processors can be indexed by the equation's subscripts to identify instances of equations assigned to individual virtual processors. Equation instances annotated by the same virtual processor constitute the smallest granule of parallel computation. An example of the use of EPL annotations in a program for the LU decomposition of a matrix is shown in Figure 2.

```
int: n;  /* array size */
real: Ain[*,*],U[*,*],L[*,*];
subscript: i,j;

range.Ain=n; range(2).Ain=n;  range.U[j]=j-1;  range.L[i]=i;

T[i,j]:A[k,i,j] =  if k==1 then Ain[i,j]
                   else  if i==Piv[k,k] then A[k-1,Piv[k,k],j]-L[i,k-1]*U[k-1,j];
                      else A[k-1,i,j]-L[i,k-1]*U[k-1,j];
D[j]:  L[j,k] =  if j==k then 1
                 else A[k,j,k]/U[k,k];
D[j]:  U[k,j] =  A[k,Piv[k,k],j];
D[i]:  Piv[k,i] = submax(abs(A[k,i,k]),i:i>=k);
```

Figure 2: LU decomposition of a matrix A in EPL

2.2 Annotations in FORTRAN

As in EPL, the notion of a virtual processor has been introduced in annotations of FORTRAN programs. FORTRAN annotations define blocks of statements associated with a virtual processor, each virtual processor defining a parallel task. Such tasks may include synchronization statements, if they encompass disjoint blocks. FORTRAN virtual processors can have subscripts associated with them to indicate repetition. An example of an annotated FORTRAN segment for the LU decomposition of a matrix is shown Figure 3. The scope of the block extends from the point of definition in the program to the statement labeled 10. In this example, a vector of virtual processors *main*, each associated with a single loop body, is defined. Blocks can also be nested in each other. Such nesting defines a hierarchy of blocks and helps in global program optimization.

Each virtual processor produces data, typically used by other virtual processors, and in turn consumes data produced by others. Performing data-dependence analysis in a style of PTRAN [12], the annotation processor can find the dependencies local to each block and data structures produced and consumed by the block. All data produced by the block are placed in the memory of the corresponding virtual processor. The created parallel tasks are extended by communication statements needed to move data. Parallel tasks associated with virtual processors at the bottom of the block hierarchy are the smallest components used in the program synthesis. An important step towards an efficient parallelization of FORTRAN programs involves an equational transformation during which the equational equivalent of the program is generated. The transformed programs obey the single assignment rule and do not

```
          PARAMETER (N = 50)
          REAL A(N,N), TEMP
          INTEGER IPIV(N)
          DO :: main 10 K = 1, N-1
            IPIV(K) = K
            DO :: pivot 20 L = K+1, N
20            IF (ABS(A(IPIV(K), K)) .LT. ABS(A(L, K)) IPIV(K) = L
            DO :: swap 30 L = K, N
              TEMP = A(K, L)
              A(K, L) = A(IPIV(K), L)
30            A(IPIV(K), L) = TEMP
            DO :: lower 40 L = K+1, N
40            A(L, K) = A(L, K) / A(K, K)
            DO :: up_update 10 L = K+1, N
              DO 10 M = K+1, N
10              A(M, L) = A(M, L) - A(M, K) * A (K, L)
          IPIV(N)=N
          STOP
          END
```

Figure 3: LU Decomposition of a matrix A in FORTRAN

contain any control statements [5]. The transformation is done in the following steps:

Reassignments Elimination: The reassigned variables are replaced by:

- vector (additional dimension) – inside loops,
- variants – in "if" branches and basic blocks.

Condition Analysis: Conditions in the transformed program are analyzed using a Sup-Inf inequality prover [4] and the Kaufl variable elimination method [8] to find pairwise equivalent or exclusive conditions.

Variable's Variants Elimination: Variable variants created in equivalent and exclusive conditions are merged into a single variable.

Additional Dimension Elimination: Memory optimization is performed to replace entire dimensions by windows of few elements for multidimensional variables [15].

The transformed FORTRAN program is then compatible with the programs produced by annotating EPL programs.

2.3 Annotation Processing

Annotation processing includes:

- creating parallel tasks defined by annotated fragments of an original program,
- declaring ports needed to interconnect created tasks into a network,
- building task communication graph that show data dependences between created tasks.

To translate the annotated program into an efficient collection of parallel tasks, it is necessary to embed a spanning tree into the tasks communication graph [11]. The following three criteria are used in selecting such an embedding:

- **Dimension nesting:** If two tasks with different dimensionalities are connected in the task communication graph, the task with more dimensions should be located lower in the spanning tree. If, for example, tasks `T[i][j]` were located above the tasks `D[j]` in the spanning tree, the addressing and creation of child tasks in `T` would involve executing an **`if-then`** statement in all $i * j$ `T` tasks.
- **Range nesting:** Whenever possible, tasks sharing the same range should be clustered together in the spanning tree. Variables that share ranges tend to appear in the same equations. Thus, clustering such variables together decreases the number of cross-process references to distributed variables.
- **Data flow:** The total communication cost of the selected spanning tree should be the smallest among all spanning trees satisfying the above two criteria.

Let G(V,E) be a task communication graph with a set of nodes V (representing processors) and a set of edges $E \subseteq V \times V$ representing communication. With each edge $e_{i,j} \in E$ we will associate the cost $c(e_{i,j})$ that represents the volume of data being sent from the processor i to the processor j. With each spanning tree T, we will also associate the distance $d^T(e_{i,j})$ that defines the minimum number of tree edges that have to be traversed on the path from task i to task j . The cost of the spanning tree T can then be defined as:

$$C(T) = \sum_{e_{i,j} \in E} c(e_{i,j}) * d^T(e_{i,j})$$

To minimize the total communication cost we need to find a proper cut-tree, which can be done by solving $| V |$ maximal flow problems. Each maximal flow problem requires $O(| V |^3)$ applications of the Ford-Fulkerson labeling procedure. Hence, finding the solution takes $O(| V |^4)$ steps.

Trees created from annotations of LU decomposition programs are shown in Figure 4 (for EPL and FORTRAN programs).

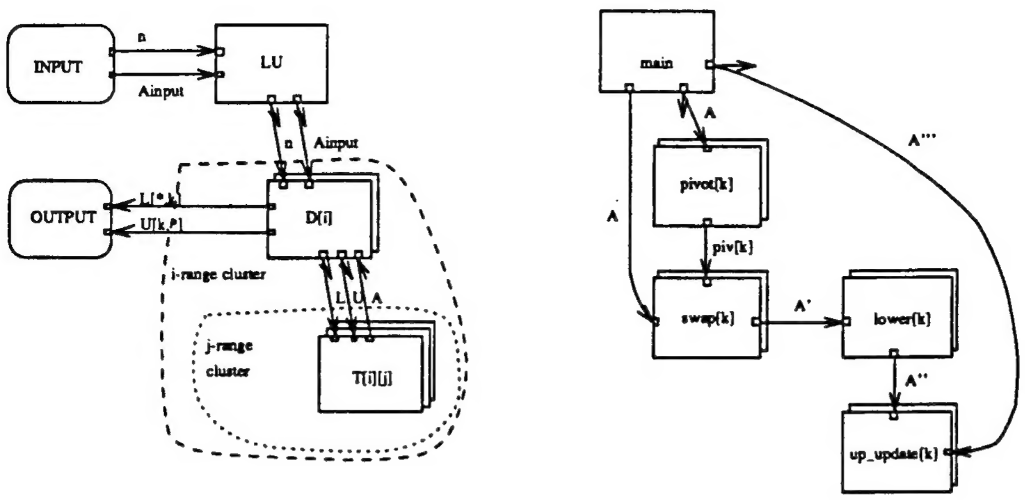

Figure 4: Communication tree for EPL and FORTRAN programs

3 Program Synthesis

In our approach a parallel computation is viewed as a collection of cooperating components. The components are defined during the program decomposition. Their cooperation requires an additional description, called a *configuration.* The configuration guides the process of synthesis. For example, components of the configuration that communicate frequently can be synthesized into a single task. The ratio of physical processors to virtual processors dictates how virtual tasks are to be mapped onto the target architecture. Usually, different annotations result in different configurations and, hence, cause different code to be generated. The user can, therefore, experiment with various annotations to find the one that results in the most efficient code. The configurator uses the dependence graph created during configuration analysis to generate an architecture-independent parallel description which is fed to the code generator.

Configurations define tasks (and their aggregates) and ports. Statements of the configuration represent relations between ports in different tasks. Some of this statements are generated during decomposition (at the subprogram level), others can be supplied by the user (when the programs are integrated into a computation).

Tasks created dynamically can communicate with ports located at parent, child, and sibling tasks (each of those tasks is just a copy of the same program or program fragment, except that a parent task can be arbitrary).

The goal of configuration processing is to establish scheduling constraints for the overall computation. In the parallel computation, individual process correctness is a necessary but not sufficient condition for the correctness of the entire computation. If a task has input/output ports that belong to a cycle in the configuration graph, then this task's input messages are dependent on the output messages. Such dependences (in addition to dependences imposed by the statements of a task) have to be taken into account in generating the object program for individual tasks; otherwise, loss of messages, process blocking, or even a deadlock can arise.

The algorithm for finding external data dependences has been presented in [13]. It produces *configuration dependence* file used by the synthesizer and the code generator. This file contains a list of the additional, externally imposed data dependences (edges and their dimension types) that need to be added to the task array graph. One task may have several such files, each associated with the different configuration in which this task participates.

4 Run-Time Task Distribution

One of the most challenging problems encountered while implementing adaptive scientific computations on distributed memory machines is run-time mapping of a dynamically changing computational load onto the parallel processors. The published solutions to this problem focus mostly on MIMD architectures and coarse grain parallelism [3]. Recently the following *Rectilinear Partitioning Problem* (RPP) has been considered in [9]: Partition the given $n \times m$ workload matrix into $(N+1) \times (M+1)$ rectangles with $N + M$ rectilinear cuts in such a way that the maximum workload among rectangles is minimized. Such optimization is appropriate for adaptive finite element computations on architectures with local communication that is faster than global. Since balanced partitions tend to increase the volume of local vs. global communication, solution to RPP decreases the overall communication costs.

In [10] we investigated adaptive scientific computations on SIMD machines, the problem with similar motivation and applications as RPP [9]. Unlike RPP however, in which the sum of the weights is taken as the cost of a rectangle, we measure the rectangular costs as the ratio of workload to the area of the rectangle that represents the number of processors active in that rectangle. Our approach is motivated by the mesh refinement techniques of the considered adaptive methods and the newly introduced coordinated parallelism on the CM-5 computer. In coordinated parallelism a machine can be partitioned into several parts each running SIMD code. The workload redistribution results in regions that have different time-step and/or grid size; therefore, the same computation is nested in loops with different boundaries. That means that each region either has to be done on the whole machine (sequentially, one after the other on the CM-2) or in a separate partition (in parallel on the CM-5). Each entry in the workload matrix represents the error in the solution obtained by an error estimation procedure [2]. The high-error regions need recomputing to the extent that is proportional to the magnitude of the error. Hence, the number of processors reassigned to each solution region should be proportional to the refinement factor.

Consider a load balancing problem as illustrated in Figure 5 for a one-dimensional

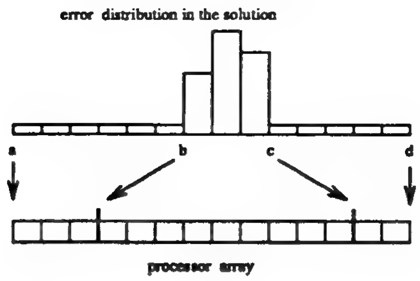

Figure 5: Example of partitioning in one-dimension

problem. The uniform mesh yields the solution with a high error in the interval $b \leq x \leq c$ and within the required accuracy in intervals $a \leq x \leq b$ and $c \leq x \leq d$. Taking the magnitude of an error as an estimate of the work ω_i for each element $i = 1, \ldots, n$, we assign a small weight $\epsilon << max_i\{\omega_i\}$ to work estimate in regions $a \leq x \leq b$ and $c \leq x \leq d$. To balance the workload, the majority of the processors should be assigned the interval $b \leq x \leq c$.

In adaptive solutions of partial differential equations parallel tasks perform basically the same computation over different spatial subdomains (intervals for one-dimensional problems) and with different discretization parameter Δx. Let K denote the number of such tasks. It is important to keep this number small for the following reasons. The subdomain interactions are proportional to the number of existing subdomains and in higher dimensions such interactions require time-consuming global communications. In each time step of the subdomain computation, a fraction of executed code is subdomain specific (e.g. in hyperbolic equations the time step has to be set differently in each subdomain). For purely SIMD machines, execution of this code fraction has to be done in K consecutive stages. In each stage, processors in one subdomain are executing while processors belonging to the remaining $K - 1$ subdomains remain idle[1]. Therefore, each subdomain associated with a parallel task should represent a localized structure in the solution domain.

Figure 6(a) shows an example of the more difficult two-dimensional case in which a coarse mesh is trivially mapped to the processor mesh. In regions A and B, the mesh must be refined due to the presence of high errors. Hence, we have to spread sub-domains A and B over bigger rectangular sub-sets of processors to improve load balancing as in Figures 6(b) and (c).

If we are employing *mesh-movement* or *static rezone* techniques, the mesh elements are moved into high-error regions. A *global* solution strategy will refine the high-error regions and repeat the entire step of the iteration. Consequently, we will need a re-assignment of processors. A *local* solution strategy, on the other hand, repeats

[1]For more general architectures, capable of coordinated parallelism mode of execution (i.e. CM-5), all K subdomains will be able to execute this fraction of code in parallel.

the iteration only where it is needed. Hence, local refinement results in less direct computation and enables more processors to be assigned to regions A and B. However, local refinement requires more interactions between the local and global solutions. Such interactions involve global communication that can outweight the benefits of an adaptive procedure. Global solutions and mesh-movement techniques require less interactions of this kind. Careful buffering of the high-error regions can increase the number of iterations executed before regridding or mesh movement is needed. This will in turn decrease the frequency of the needed load balancing. It is this global mesh-refinement and mesh movement techniques executed on a mesh connected architectures that motivated us to develop density-type partitioning.

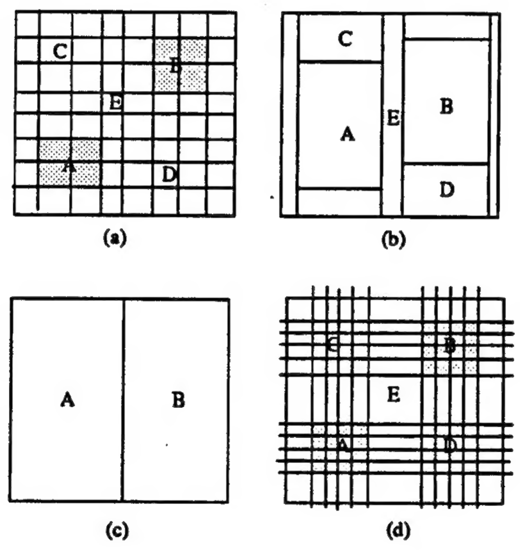

Figure 6: (a) Coarse mesh with high error regions A and B, (b) repartitioning with global refinement (c) repartitioning with local refinement (d) Nicol's partitioning

It should be noted that applying Nicol's [9] partitioning methodology RPP to the example shown in Figure 6(d) results in assigning unnecessary processors to regions C and D. To avoid such waste, we did not restrict our partitioning methodology to rectilinear cuts extending across the whole domain in both dimensions. Instead, in our problem definition and solution [10], we require that K selected rectangles cover the whole domain. The heuristics for the two-dimensional case projects the weights to one-dimension and results in rectilinear cuts extending across the whole dimension in one direction. Figure 6(b) shows an example of this kind of partition.

Let P_K be a set of partitions of a one-dimensional workload array ω_i, $i = 1, \ldots, n$ into K subintervals (x_{1_k}, x_{2_k}), where $1 \leq x_{1_k} \leq x_{2_k} \leq n$, $k = 1, \ldots, K$. The one-dimensional workload partitioning problem can be then stated as:

$$\bigoplus \left\{ \bigotimes_k \left\{ \frac{\sum_{i=x_{1_k}}^{x_{2_k}} \omega_i}{f(x_{1_k}, x_{2_k})} \right\} : (x_{1_k}, x_{2_k}) \quad k = 1, \ldots, K \in P_K \right\} \tag{1}$$

As shown in Table 1, selecting different meaning for operations $\bigoplus$ and $\bigotimes$ we can obtain different optimization problems from this formulation. For $\bigoplus \equiv min$, $\bigotimes \equiv max$ and $f(x_{1_k}, x_{2_k}) = 1$ we obtain the Nicol's problem that has solutions of complexity $O(Kn)$ and $O(n + (Klogn)^2)$ [9].

Problem	$\bigoplus$	$\bigotimes$	$f(x_{1_k}, x_{2_k})$	ϵ	e
Nicol's 1D partitioning	min	max	1	∞	0
Density-type for PDEs	min	max	$(x_{2_k} - x_{1_k} + 1)$	∞	0
Shortest path with k arcs	min	+	1	∞	0
Density-type for PDEs	max	min	$(x_{2_k} - x_{1_k} + 1)$	0	∞

Table 1: *Instances of problem represented by equation (1)*

The problem involving load balancing for adaptive PDE solvers discussed in this section is obtained for $\bigoplus \equiv min$, $\bigotimes \equiv max$ and $f(x_{1_k}, x_{2_k}) = (x_{2_k} - x_{1_k} + 1)$, i.e., we divide the sum of the workloads in each partition by the interval length (i.e., the number of processors). There is a similarity between the weighted independent set for interval graphs and our problem [7]. The interval graph for our problem can be created by having a node representing one of the possible subintervals (x_{1_k}, x_{2_k}) with the weight $\sum_{i=x_{1_k}}^{x_{2_k}} \omega_i / f(x_{1_k}, x_{2_k})$ and edges representing the intersections between the subintervals. In such a graph, the independent set of size K which covers the whole interval, $1, \ldots, n$, gives the solution to the original problem. We convert that interval graph to a directed acyclic graph (DAG) and apply the shortest path algorithm to find the minimum weight dominating set [10]. This approach results in the optimal algorithm for the one-dimensional case and leads also to a heuristic algorithm that can be easily generalized to two dimensions (by projecting the workloads to one dimension).

5 Conclusion

Our approach is based on the following presumptions:

- Adaptivity is at the center of efficient methods for solving partial differential equations.

- Annotations provide an easy and efficient way for parallelization of existing codes.

- Absence of control statements simplifies program analysis and increases the compiler ability to produce an efficient parallel code.

- Most parallel code optimization problems are NP-hard; hence, development of proper heuristics is important.

- A hierarchical view of parallel computation is helpful in extracting functional parallelism.

Program decomposition through annotations and computation synthesis through configuration can support efficient parallel code generation for domain-specific computation. Adaptivity, with its associated error estimates and shrewd use of computation only in regions where accuracy requirements are not satisfied, can provides the needed numerical reliability and efficiency to parallel computation. Massive parallelism combined with adaptivity offers a promise of true breakthroughs that will allow scientists and engineers to solve the most demanding problems with available resources.

Our research on scalable program synthesis is in its early stages and many issues remain unexplored. Future work on program synthesis should include more work on run-time code optimization. Large applications will measure the efficiency of the generated solutions.

References

[1] Baber, M.: The Hypertasking Paracompiler - Parallelizing the Game of Life and Other Applications. Supercomputing Review. 3, 41–47 (1991)

[2] Flaherty, J. E., Paslow, P. J., Shephard, M.S. and Vasilakis, J. D., (eds) Adaptive Methods for Partial Differential Equations, SIAM, Philadelphia, 1989.

[3] Berger, M.J., and Bokhari, S.H.: A Partitioning Strategy for Nonuniform Problems on Multiprocessors. IEEE Trans. on Computers. C-36, 570–580 (1987)

[4] Bruno, J., and Szymanski, B.K.: Analyzing Conditional Data Dependencies in an Equational Language Compiler. Proc. 3rd Supercomputing Conference 1988, Boston, MA, pp. 358–365. Tampa. FL: Supercomputing Institute 1988

[5] Ge X., and Prywes, N.S.: Reverse Software Enginnering of Concurrent Programs. Proc. 5th Jerusalem Conference on Information Technology 1990, Jerusalem, pp. 731–742. Washington, DC: IEEE Computer Science Press 1990

[6] Gelernter, D., and Carriero, N.: Coordination Languages and their Significance. Comm. ACM. 35, 97–107 (1992)

[7] Golumbic, M.C.: Algorithmic Graph Theory and Perfect Graphs. New York. NY: Academic Press 1980

[8] Kaufl, T.: Reasoning about Systems of Linear Inequalities. In: Ninth International Conference on Automated Deduction. Aragon. IL, Lecture Notes in Computer Science, pp. 563–72. Berlin-Heidelberg-New York: Springer 1988

[9] Nicol, D.M.: Rectilinear Partitioning of Irregular Data Parallel Computations. ICASE NASA, Report 91-55, 1991

[10] Özturan, C., Szymanski, B.K., and Flaherty, J.: Adaptive Methods and Rectangular Partitioning Problem. Proc. Scalable High Performance Computing Conference 1992, Wilmington. VA, pp. 409–415. Washington. DC: IEEE Computer Society Press 1992

[11] Özturan, C.: Expressing Parallelism in EPL. Rensselaer Polytechnic Institute, Tech. Report No. 90-29, December 1990

[12] Sarkar, V.: PTRAN - The IBM Parallel Translation System," In: Parallel Functional Languages and Compilers (B.K. Szymanski, ed.). pp. 309–391. New York. NY: ACM Press 1991

[13] Spier, K., and Szymanski, B.K.: Interprocess Analysis and Optimization in the Equational Language Compiler. In: CONPAR-90. Lecture Notes in Computer Science, pp. 287–98. Berlin-Heidelberg-New York: Springer 1990

[14] Szymanski, B.K.: EPL - Parallel Programming with Recurrent Equations. In: Parallel Functional Languages and Environments (B.K. Szymanski ed.). pp. 51–104. New York. NY: ACM Press, 1991

[15] Szymanski, B.K., and Prywes, N.S.: Efficient Handling of Data Structures in Definitional Languages. Science of Computer Programming. 10, pp. 221–245 (1988)

[16] Walker, T.M. The Federal High Performance Computing Program. Comput. Res. News 1, (1989).